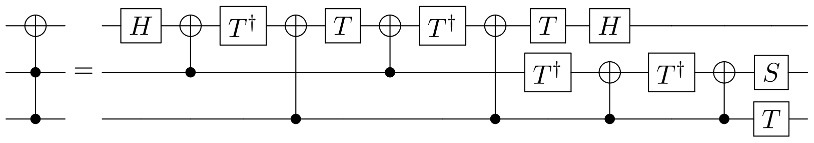
Chapter 7
Quantum Algorithms

In Chapters 1 and 4, we saw a classical algorithm and a quantum algorithm for adding two binary numbers, each of length n. The classical ripple-carry adder in Section 1.3.4 used $5n - 3$ logic gates, whereas from Section 4.5.7, the quantum ripple-carry adder used $4n - 2$ Toffoli gates and $4n$ CNOT gates, for a total of $8n - 2$ gates. Thus, the quantum algorithm uses more gates than the classical algorithm. Furthermore, if we decompose the Toffoli gates into one- and two-qubit gates, the number of quantum gates would be even greater. This is a disappointing result. We want quantum computers to be faster (i.e., use fewer gates) than classical computers. In this chapter, we consider quantum algorithms that are actually better than their classical counterparts.

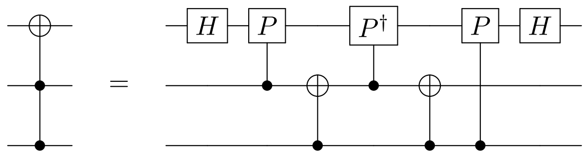
7.1 Circuit vs Query Complexity

7.1.1 Circuit Complexity

The most precise way to quantify the complexity of a quantum circuit is to count the least number of quantum gates required to implement it, relative to some universal set of quantum gates. This is called its *circuit complexity*. For example, if we permit only one- and two-qubit quantum gates, then recall from Exercise 4.23 that the Toffoli gate can be decomposed into



This has sixteen one- and two-qubit gates, but it is not the circuit complexity of the Toffoli gate. In the top row, the last T and H gates can be combined into a single one-qubit gate, reducing the number of one- and two-qubit gates to fifteen. Yet this is still not the circuit complexity. A circuit that uses even fewer one- and two-qubit gates is



where P is some one-qubit gate. This only uses seven one- and two-qubit gates. It is an active area of research to determine whether Toffoli can be simplified further. If only CNOT and one-qubit gates are permitted, however, it has been proved that Toffoli requires at least six CNOT gates (plus one-qubit gates). Suffice it to say that circuit complexity is generally hard to find.

In terms of circuit complexity, an *efficient* quantum algorithm is one with a polynomial circuit complexity. For example, the quantum adder is efficient since its $4n-2$ Toffoli gates can each be decomposed into seven one- and two-qubit gates. Adding the $4n$ CNOT gates, this results in $7(4n-2)+4n = 32n-14$ one- and two-qubit gates, which is polynomial (linear) in n.

It is also difficult to find the circuit complexity of classical circuits. It can be hard to determine if a logic circuit has been fully simplified, and it depends on what gates are allowed. For example, should the final circuit only consist of AND, OR, and NOT, or can it also include XOR? Should it only consist of NAND gates? Are three-bit logic gates allowed, or only one- and two-bit gates?

7.1.2 Query Complexity

Since circuit complexity can be hard to find, we often turn to *query complexity* instead. The query complexity of a problem is the number of calls to a function, or *queries* to an *oracle* or *black box* needed to solve the problem. We give an input to the function or oracle or black box, and it returns an output, without us knowing its inner workings. Hence, it is opaque or black. It is significantly easier to find the query complexity of a problem, or when that is not possible, mathematically prove upper or lower bounds on it.

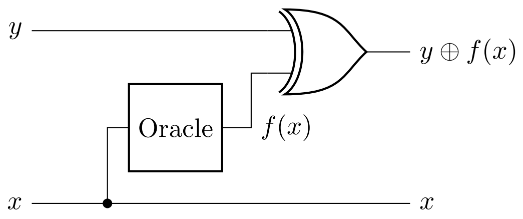
For example, a problem we will explore later this chapter is brute-force searching. Say we are searching a database of 100 items for one particular item, and we have an oracle that tells us whether an item is the correct one or not. We can query, "Oracle, is item number 1 correct?" If the oracle replies, "Yes," we have found our item. If the oracle replies "No," we can inquire about another item: "Oracle, is item number 2 correct?" We can continue in this manner until the oracle says "Yes." Mathematically, the oracle is just a function $f(x)$ that outputs 0 (no) or 1 (yes). So, evaluating $f(1)$ is inquiring whether item number 1 is correct. If $f(1) = 1$, we have found the correct item, but if $f(1) = 0$, we can inquire about another item, like $f(2)$, and so on. The query complexity is the number of times we need to evaluate $f(x)$

in order to find the item. As we will see later in Section 7.6, if the database has N entries, a classical computer takes $O(N)$ queries, but a quantum computer only takes $O(\sqrt{N})$ queries using Grover's algorithm. We call such an improvement or speedup in the number of oracle queries an *oracle separation*.

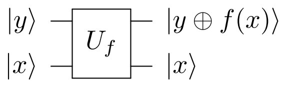
The first part of this chapter will cover quantum algorithms with oracle separations, meaning they take fewer queries than classical computers to solve problems. These algorithms are generally easier to understand. Then, the second part of this chapter will cover quantum algorithms with better circuit complexities, which are generally more advanced. Before we start looking at oracular problems, i.e., problems with an oracle, let us discuss next how an oracle $f(x)$ acts in a quantum computer.

7.1.3 Quantum Oracles

An oracle is simply a *boolean function*, meaning a function that acts on bits. Then, it can be defined using a truth table, and it can be constructed using logic gates. For it to be a quantum oracle, however, it needs to be reversible. Fortunately, from Section 1.5.4, we can turn it into a reversible circuit by XORing its output with an extra bit. For example, if the function is $f(x)$, the reversible circuit is



Since this entire circuit is reversible, it is a quantum gate. Let us call the gate U_f to emphasize that it is unitary. We can draw it as



That is, the quantum oracle U_f acts as

$$|x\rangle|y\rangle \xrightarrow{U_f} |x\rangle|y \oplus f(x)\rangle.$$

Note we can find $f(0)$ and $f(1)$ by setting $y = 0$. That is,

$$\begin{aligned} |0\rangle|0\rangle &\xrightarrow{U_f} |0\rangle|0 \oplus f(0)\rangle = |0\rangle|f(0)\rangle, \\ |1\rangle|0\rangle &\xrightarrow{U_f} |1\rangle|0 \oplus f(1)\rangle = |1\rangle|f(1)\rangle. \end{aligned}$$

The extra qubit $|y\rangle$ is called an *answer qubit* or *target qubit*, and $|x\rangle$ is called the *input qubit*.

Exercise 7.1. Consider a classical oracle $f(x) = \bar{x}$, where x is a bit, and $\bar{x}$ is the NOT of x. We want to turn this into a quantum oracle U_f that acts according to

$$|x\rangle|y\rangle \xrightarrow{U_f} |x\rangle|y \oplus f(x)\rangle.$$

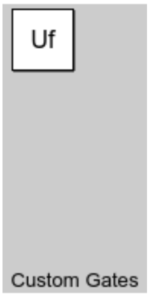
Answer the following questions about this operator:

(a) What is the truth table of the quantum oracle?

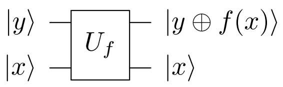
x	y	x	$y \oplus f(x)$
0	0	?	?
0	1	?	?
1	0	?	?
1	1	?	?

(b) Is the operation reversible?
(c) What is the quantum oracle as a 4×4 matrix?
(d) Verify that the matrix is unitary.

Exercise 7.2. Go to https://bit.ly/3m3Zcei. By following this link, you should have access to a custom gate called U_f.



This is the oracle. It acts on two qubits according to



(a) Using Quirk, query the oracle with appropriate inputs to find $f(0)$.
(b) Using Quirk, query the oracle with appropriate inputs to find $f(1)$.

7.1.4 Phase Oracle

If we query a quantum oracle the standard way described above, the input qubit $|x\rangle$ is unchanged while the answer qubit $|y\rangle$ becomes $|y \oplus f(x)\rangle$. There is a way to query the quantum oracle, however, that causes the answer qubit $|y\rangle$ to be unchanged while multiplying $|x\rangle$ by a phase. It works by setting $|y\rangle = |-\rangle$, which can be done by initializing the answer qubit to $|0\rangle$, applying X to turn it into $|1\rangle$, and then applying H to turn it into $|-\rangle$. That is, writing both the input and answer qubits,

$$|x\rangle|0\rangle \xrightarrow{I \otimes X} |x\rangle|1\rangle \xrightarrow{I \otimes H} |x\rangle|-\rangle.$$

Note it is possible to prepare the answer qubit $|y\rangle$ in the state $|-\rangle$ because the state of a qubit can be a superposition of $|0\rangle$ and $|1\rangle$; with a classical answer bit, this would be impossible. Now, let us expand $|x\rangle|-\rangle$ and see what happens when we query the oracle:

$$\begin{aligned}
|x\rangle|-\rangle &= |x\rangle\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\
&= \frac{1}{\sqrt{2}}(|x\rangle|0\rangle - |x\rangle|1\rangle) \\
&\xrightarrow{U_f} \frac{1}{\sqrt{2}}(|x\rangle|0\oplus f(x)\rangle - |x\rangle|1\oplus f(x)\rangle) \\
&= \begin{cases} \frac{1}{\sqrt{2}}(|x\rangle|0\rangle - |x\rangle|1\rangle), & f(x)=0 \\ \frac{1}{\sqrt{2}}(|x\rangle|1\rangle - |x\rangle|0\rangle), & f(x)=1 \end{cases} \\
&= \begin{cases} |x\rangle|-\rangle, & f(x)=0 \\ -|x\rangle|-\rangle, & f(x)=1 \end{cases} \\
&= (-1)^{f(x)}|x\rangle|-\rangle.
\end{aligned}$$

We can interpret this as the answer qubit staying in the $|-\rangle$ state while the input qubit goes from $|x\rangle$ to $(-1)^{f(x)}|x\rangle$. That is, the input qubit acquires a phase. This is called *phase kickback*. Often, we drop the answer qubit, since it stays in the $|-\rangle$ state, and only write the input qubit:

$$|x\rangle \xrightarrow{U_f} (-1)^{f(x)}|x\rangle.$$

This is called a *phase oracle*, where the qubit $|x\rangle$ is multiplied by a phase $(-1)^{f(x)}$. The phase oracle will be very useful for the oracular problems we will cover.

Exercise 7.3. Quantum oracles are quantum gates, so they act across superpositions. Consider an input qubit in the superposition state

$$\frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle,$$

and an answer qubit in the state $|-\rangle$. Show that the quantum oracle acts by

$$\left(\frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle\right)|-\rangle \xrightarrow{U_f} \left(\frac{\sqrt{3}}{2}(-1)^{f(0)}|0\rangle + \frac{1}{2}(-1)^{f(1)}|1\rangle\right)|-\rangle.$$

Exercise 7.4. We saw that when the answer qubit is in the state $|-\rangle$, we get phase kickback. Let us explore what happens if the answer qubit is in the state $|+\rangle$. Suppose an input qubit and answer qubit are in the state $|x\rangle|+\rangle$, where x is a bit. If we apply the quantum oracle U_f to this, which maps $|x\rangle|y\rangle$ to $|x\rangle|y\oplus f(x)\rangle$, what do we get if
(a) $f(x)=0$?
(b) $f(x)=1$?
(c) How do your answers in parts (a) and (b) compare to the initial state $|x\rangle|+\rangle$?

7.2 Parity

7.2.1 *The Problem*

For the first algorithm, we have two unknown bits b_0 and b_1, and we want to find the parity of the two bits. That is, we want to find $b_0 \oplus b_1$, or equivalently, whether the number of 1's is even or odd. To do this, we are given an oracle $f(x) = b_x$ that takes as input an index $x \in \{0,1\}$ and returns the corresponding bit. That is,

$$f(0) = b_0, \quad f(1) = b_1.$$

We will show that to find the parity of b_0 and b_1, we must query this oracle twice classically, but only once quantumly.

7.2.2 *Classical Solution*

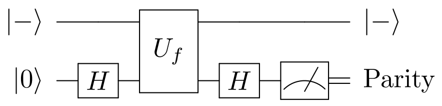
Classically, we need to know both bits in order to find $b_0 \otimes b_1$. So, we need to query the oracle twice, once to find b_0 and again to find b_1:

$$b_0 = f(0), \quad b_1 = f(1).$$

Then we take the XOR of b_0 and b_1 to find the parity. Thus, the classical query complexity is 2.

7.2.3 *Quantum Solution: Deutsch's Algorithm*

Quantumly, it only takes 1 query using *Deutsch's algorithm.* It uses one input qubit and one answer qubit, and the algorithm is shown in the following quantum circuit:



Let us work through each step of the circuit. Ignoring the answer qubit, we first apply the Hadamard gate to the input qubit:

$$|0\rangle \xrightarrow{H} \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle).$$

Next, we query the oracle, which acts as a phase oracle because the answer qubit is $|-\rangle$. From Exercise 7.3, we get

$$\frac{1}{\sqrt{2}}\left[(-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle\right].$$

To show why this is helpful, let us rewrite this. First, we substitute $f(0) = b_0$ and $f(1) = b_1$:

$$\frac{1}{\sqrt{2}}\left[(-1)^{b_0}|0\rangle + (-1)^{b_1}|1\rangle\right].$$

Then, we factor out $(-1)^{b_0}$.

$$(-1)^{b_0}\frac{1}{\sqrt{2}}\left[|0\rangle + (-1)^{b_1-b_0}|1\rangle\right].$$

Now, depending on whether b_0 and b_1 are equal, this is

$$\begin{cases} (-1)^{b_0}\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), & b_0 = b_1 \\ (-1)^{b_0}\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle), & b_0 \neq b_1 \end{cases}.$$

These are just $|+\rangle$ and $|-\rangle$, each with a phase:

$$\begin{cases} (-1)^{b_0}|+\rangle, & b_0 = b_1 \\ (-1)^{b_0}|-\rangle, & b_0 \neq b_1 \end{cases}.$$

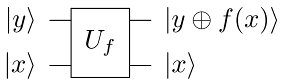
So, if b_0 and b_1 are equal, we have $|+\rangle$ with an overall phase, and if b_0 and b_1 are unequal, we have $|-\rangle$ with an overall phase. We can distinguish these by measuring in the X-basis $\{|+\rangle, |-\rangle\}$. Or, we can apply the Hadamard gate again and then measure in the Z-basis $\{|0\rangle, |1\rangle\}$. Applying the Hadamard gate, we get

$$\begin{cases} (-1)^{b_0}|0\rangle, & b_0 = b_1 \\ (-1)^{b_0}|1\rangle, & b_0 \neq b_1 \end{cases}.$$

If we measure this, we either get $|0\rangle$ or $|1\rangle$, since the overall phase of $(-1)^{b_0}$ does not matter. If we get $|0\rangle$, we know that $b_0 = b_1$, and so the parity of the bits is 0 (even). On the other hand, if we get $|1\rangle$, we know that $b_0 \neq b_1$, and so the parity of the bits is 1 (odd). Thus, depending on whether we get $|0\rangle$ or $|1\rangle$, we know whether the parity of the two bits is 0 or 1, and we determined this with just one query to the oracle. Thus, the quantum query complexity is 1, which is an improvement over the classical query complexity of 2. While the improvement in query complexity from 2 to 1 may be small, it is our first concrete example of a quantum computer outperforming a classical computer.

Note in Deutsch's algorithm, we never learned the values of b_0 and b_1 themselves, which would require two oracle queries. Instead, we only learned whether they are equal or opposite, which corresponds to even or odd parity, respectively.

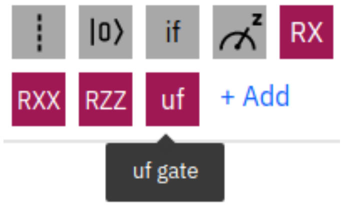
Exercise 7.5. There are two unknown bits b_0 and b_1, and you want to find the parity of the two bits by querying an oracle. Go to `https://bit.ly/2ILe3cF`. By following this link, you should have access to a custom gate called U_f. This is the oracle, and it acts on two qubits by



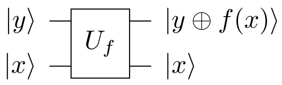
For this problem, the function $f(x)$ returns bit x, so $f(x) = b_x$.

(a) In Quirk, use Deutsch's algorithm to find the parity of b_0 and b_1 using just one query to U_f. Note you will need to prepare the answer qubit so that it is in the minus state.
(b) In Quirk, query U_f in such a way as to find b_0.
(c) In Quirk, query U_f in such a way as to find b_1.
(d) Since you now know b_0 and b_1 from parts (b) and (c), find their parity. Verify that it agrees with your result from part (a).
(e) In the worst case, how many queries does it take to solve the problem classically?

Exercise 7.6. There are two unknown bits b_0 and b_1, and you want to find the parity of the two bits by querying an oracle. Go to `https://ibm.co/3GxSWTT`. By following this link, you should have access to a custom gate called U_f:



This is the oracle, and it acts on two qubits by



For this problem, the function $f(x)$ returns bit x, so $f(x) = b_x$.

(a) Program Deutsch's algorithm in IBM Quantum, and use the quantum simulator to find the parity of b_0 and b_1.
(b) Run the circuit on an actual quantum processor using IBM Quantum. Which processor did you use, and what histogram of results do you get?

Exercise 7.7. Say you are trying to use Deutsch's algorithm, but you neglect the last Hadamard gate. That is, you apply

$$|0\rangle \xrightarrow{H} |+\rangle \xrightarrow{U_f} \frac{1}{\sqrt{2}}\left[(-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle\right].$$

If you measure the system now, what possible states do you get, and with what probabilities?

7.2.4 *Generalization to Additional Bits*

What if we have n bits $b_0, b_1, \ldots, b_{n-1}$, and we want to find their parity? Classically, we need to know all n bits, so it takes n queries. Quantumly, we can use Deutsch's algorithm to find the parity of pairs of bits:

$$\underbrace{b_0, b_1}_{\text{parity}}, \underbrace{b_2, b_3}_{\text{parity}}, \ldots \underbrace{b_{n-2}, b_{n-1}}_{\text{parity}}.$$

This takes $n/2$ queries. Then we can take the XOR of all these parities to get the parity of all the bits. This takes no additional queries. So, the quantum query complexity is $n/2$, which is half classical query complexity. Note both the classical and quantum runtimes are $O(n)$, however, so there is no improvement in their asymptotic scaling.

Exercise 7.8. You have eight bits, and using Deutsch's algorithm, you have found the parities of pairs of bits, shown below:

$$\underbrace{b_0, b_1}_{1}, \underbrace{b_2, b_3}_{1}, \underbrace{b_4, b_5}_{0}, \underbrace{b_6, b_7}_{1}.$$

(a) What is the parity of all eight bits?
(b) How many queries to the oracle did it take to find the parity of all eight bits?
(c) In the worst case, how many queries does it take to solve the problem classically?

Exercise 7.9. You have nine bits, and using Deutsch's algorithm, you have found the parities of the first four pairs of bits. Then you queried the oracle for the last bit, revealing whether it's a 1 or a 0. This is shown below:

$$\underbrace{b_0, b_1}_{0}, \underbrace{b_2, b_3}_{1}, \underbrace{b_4, b_5}_{0}, \underbrace{b_6, b_7}_{0}, \underbrace{b_8}_{1}.$$

(a) What is the parity of all nine bits?
(b) How many queries to the oracle did it take to find the parity of all nine bits?
(c) In the worst case, how many queries does it take to solve the problem classically?

7.3 Constant vs Balanced Functions

7.3.1 The Problem

In this problem, we have a function $f(x)$ that takes as input a binary number $x = x_{n-1} \ldots x_1 x_0$ of length n and outputs 0 or 1. Mathematically, we can write this as $f : \{0,1\}^n \to \{0,1\}$, where $\{0,1\}^n$ denotes bit strings of length n. We additionally have the promise that f is constant (always outputs 0 or always outputs 1) or *balanced* (outputs 0 half the time, and outputs 1 half the time), and the problem is to determine which we have. Put another way, f outputs 1 none of the time, all of the time, or half of the time, and the task is to determine if it is none or all of the time (constant) or half of the time (balanced).

For example, the following function on binary strings of length 3 (i.e., on 3 bits) is balanced since it outputs 0 half the time and 1 the other half of the time:

x_2	x_1	x_0	$f(x)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Note $n = 1$ is Deutsch's algorithm since the input is a single bit, and if the function is constant, the parity is 0 or even, and if the function is balanced, the parity is 1 or odd. So the Deutsch-Jozsa algorithm can be seen as a generalization of Deutsch's algorithm.

7.3.2 Classical Solution

Classically, to determine with certainty whether f is constant or balanced, we need to query half the inputs, plus one, in the worst case scenario. That is, if we query half the inputs and get zero each time, then we still do not know if just half the outputs are zero, or if all the outputs are zero. Querying one more input resolves this. Since there are 2^n possible inputs (binary strings of length n), the classical query complexity is $2^{n-1} + 1$. Note this scales exponentially in n, i.e., it is $O(2^n)$.

In practice, however, one may accept a classical algorithm that guesses the correct answer most of the time. For example, say we query f with $c = 10$ different random inputs, and we get $f = 0$ each time. Then, we can guess that f is constant with some degree of certainty. As we will show next, the probability that our guess is wrong can be made smaller than any constant using some suitable constant value for c. Then, such a randomized algorithm can solve the problem with just c queries, which is $O(1)$.

To show this, say we classically query f for c different random inputs. If we get the same output every time, we guess that f is constant, and if we get a mix of 0's and 1's, we guess that f is balanced. Let us calculate the probability that these guesses are incorrect. First, if f is constant, we will get the same output for all c of our inputs, and we will correctly guess that f is constant, so there is no error in this case. If f is balanced and the c outputs are any mix of 0's and 1's, we will correctly guess that f is balanced, so there is no error in this case, either. If f is balanced and all c of our outputs are the same, however, we will incorrectly guess that f is constant, which is an error. Let us find the probability of this error. Say all c of our outputs are 0, but f is actually balanced. To get the first 0, there are $2^n/2 = 2^{n-1}$ outputs that are 0 out of a total of 2^n outputs, so the probability of getting a 0 is $2^{n-1}/2^n$. For the second 0, there are $2^{n-1} - 1$ outputs remaining that are 0 out of a total of $2^n - 1$ outputs remaining, so the probability of getting a second 0 is $(2^{n-1} - 1)/(2^n - 1)$. Continuing this reasoning, the probability of getting c zeros is

$$\begin{aligned}
&\frac{2^{n-1}}{2^n}\frac{2^{n-1}-1}{2^n-1}\frac{2^{n-1}-2}{2^n-2}\cdots\frac{2^{n-1}-(c-1)}{2^n-(c-1)} \\
&\approx \frac{2^{n-1}}{2^n}\frac{2^{n-1}}{2^n}\frac{2^{n-1}}{2^n}\cdots\frac{2^{n-1}}{2^n} \\
&= \frac{1}{2}\frac{1}{2}\frac{1}{2}\cdots\frac{1}{2} = \frac{1}{2^c},
\end{aligned}$$

where in the second line, we have approximated the expression for large n. Similarly, the probability of all c queries yielding 1 even though f is balanced is also $1/2^c$. Together, the total probability of incorrectly guessing that f is constant when it is actually balanced is

$$\frac{1}{2^c} + \frac{1}{2^c} = \frac{1}{2^{c-1}}.$$

This does not depend on n. So, we can bound the error by checking an appropriate number of inputs. For example, if we want the probability of error to be less than 1%, we only need to query f for $c = 8$ different inputs, and this is a constant number of queries regardless of n. Thus, this randomized algorithm takes $O(1)$ queries of f.

From Exercise 1.53, problems that are efficiently solved with bounded error by such randomized algorithms are contained in the complexity class *bounded-error probabilistic polynomial time* (BPP). It is believed that $P = BPP$, but it is not proven. Since we gave an efficient randomized algorithm for determining whether f is constant or balanced, this problem is in BPP.

To review, for a classical computer to determine with certainty whether f is constant or balanced, it needs $2^{n-1} + 1$ queries to f in the worst case, which is $O(2^n)$. For a probabilistic classical computer to guess the answer with bounded error, it only needs a constant number of queries to f, which is $O(1)$.

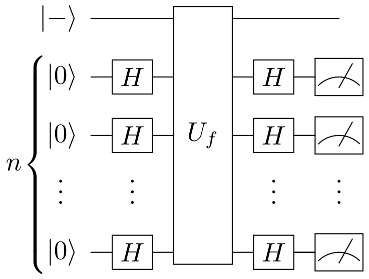
Exercise 7.10. When determining if an oracle f is constant or balanced,
(a) What is the probability of an error if you evaluate f for $c = 7$ different inputs?
(b) What is the probability of an error if you evaluate f for $c = 8$ different inputs?
(c) How many times should f be evaluated to reduce the error probability to less than 0.1%?

7.3.3 Quantum Solution: Deutsch-Jozsa Algorithm

A quantum computer using the *Deutsch-Jozsa algorithm* can determine with certainty whether f is constant or balanced using just 1 query to f. This is an exponential speedup over the exact classical algorithm, but no speedup over the bounded-error probabilistic classical algorithm.

The Deutsch-Jozsa algorithm is very similar to Deutsch's algorithm, but we now have n qubits (plus an answer qubit, which we ignore by using a phase oracle). These n qubits are initially each in the $|0\rangle$ state, and we apply Hadamards to put them in a superposition of all n-bit strings. Then, we query the oracle on this superposition.

Finally, we apply Hadamards to all the qubits to create a state that we measure, and whose measurement outcome allows us to distinguish whether the function is constant or balanced. Including the answer qubit, the Deutsch-Jozsa algorithm as a quantum circuit is



Let us work out the math to show why this determines whether f is constant or balanced. Ignoring the answer qubit, we begin with n qubits, all in the $|0\rangle$ state. First applying the Hadamard gate to each of these n qubits, we get

$$\begin{aligned}|0\rangle^{\otimes n} &\xrightarrow{H^{\otimes n}} |+\rangle^{\otimes n} \\ &= \frac{1}{\sqrt{2^n}}(|0\rangle+|1\rangle)^{\otimes n} \\ &= \frac{1}{\sqrt{2^n}}\sum_{x\in\{0,1\}^n}|x\rangle. \end{aligned} \tag{7.1}$$

So, applying Hadamards to the all zero state creates a uniform superposition over all binary strings. Next, we query the phase oracle:

$$\frac{1}{\sqrt{2^n}}\sum_{x\in\{0,1\}^n}|x\rangle \xrightarrow{U_f} \frac{1}{\sqrt{2^n}}\sum_{x\in\{0,1\}^n}(-1)^{f(x)}|x\rangle.$$

Finally, we again apply the Hadamard gate to each of the n qubits:

$$\frac{1}{\sqrt{2^n}}\sum_{x\in\{0,1\}^n}(-1)^{f(x)}|x\rangle \xrightarrow{H^{\otimes n}} \frac{1}{\sqrt{2^n}}\sum_{x\in\{0,1\}^n}(-1)^{f(x)}H^{\otimes n}|x\rangle. \tag{7.2}$$

This is the final state of the algorithm before the measurement, and to interpret this, let us focus on just one $H^{\otimes n}|x\rangle$, where $|x\rangle$ is a single n-bit string:

$$\begin{aligned}H^{\otimes n}|x\rangle &= H^{\otimes n}|x_{n-1}\ldots x_1x_0\rangle \\ &= H|x_{n-1}\rangle\ldots H|x_1\rangle H|x_0\rangle.\end{aligned}$$

Depending on whether x_i is 0 or 1, $H|x_i\rangle$ is either $|+\rangle$ or $|-\rangle$. To account for the difference in sign between $|+\rangle$ and $|-\rangle$, we can write $H|x_i\rangle$ as

$$H|x_i\rangle = \frac{1}{\sqrt{2}}\Big[|0\rangle+(-1)^{x_i}|1\rangle\Big].$$

This way, when $x_i = 0$, $(-1)^{x_i} = 1$, and the result is $|+\rangle$, and when $x_i = 1$, $(-1)^{x_i} = -1$, and the result is $|-\rangle$. Writing each $H|x_i\rangle$ like this, we get

$$\begin{aligned} &H|x_0\rangle H|x_1\rangle \dots H|x_{n-1}\rangle \\ &= \frac{1}{\sqrt{2}}\Big[|0\rangle + (-1)^{x_{n-1}}|1\rangle\Big] \dots \frac{1}{\sqrt{2}}\Big[|0\rangle + (-1)^{x_1}|1\rangle\Big] \frac{1}{\sqrt{2}}\Big[|0\rangle + (-1)^{x_0}|1\rangle\Big]. \end{aligned}$$

Multiplying out the terms, this becomes

$$\begin{aligned} \frac{1}{\sqrt{2^n}}\Big[&|0\dots000\rangle + (-1)^{x_0}|0\dots001\rangle + (-1)^{x_1}|0\dots010\rangle \\ &+ (-1)^{x_1}(-1)^{x_0}|0\dots011\rangle + (-1)^{x_2}|0\dots100\rangle + (-1)^{x_2}(-1)^{x_0}|0\dots101\rangle \\ &+ (-1)^{x_2}(-1)^{x_1}|0\dots110\rangle + (-1)^{x_2}(-1)^{x_1}(-1)^{x_0}|0\dots111\rangle + \dots\Big]. \end{aligned}$$

Writing $(-1)^{x_1}(-1)^{x_0}$ as $(-1)^{x_1+x_0}$, and similarly elsewhere, we get

$$\begin{aligned} \frac{1}{\sqrt{2^n}}\Big[&|0\dots000\rangle + (-1)^{x_0}|0\dots001\rangle + (-1)^{x_1}|0\dots010\rangle \\ &+ (-1)^{x_1+x_0}|0\dots011\rangle + (-1)^{x_2}|0\dots100\rangle + (-1)^{x_2+x_0}|0\dots101\rangle \\ &+ (-1)^{x_2+x_1}|0\dots110\rangle + (-1)^{x_2+x_1+x_0}|0\dots111\rangle + \dots\Big]. \end{aligned}$$

This is a sum over all n-bit strings $|z\rangle = |z_{n-1}\dots z_1 z_0\rangle$, so it becomes

$$\frac{1}{\sqrt{2^n}} \sum_{z\in\{0,1\}^n} (-1)^{\sum_{i:z_i=1} x_i}|z\rangle.$$

For the negative sign, the power is the sum of the values of x_i such that $z_i = 1$. We can also write this sum using the *dot product* $x \cdot z$,

$$x \cdot z = x_{n-1}z_{n-1} + \dots + x_1 z_1 + x_0 z_0.$$

In this dot product, the only x_i's that survive are those where $z_i = 1$. Using this notation, we get

$$H^{\otimes n}|x\rangle = \frac{1}{\sqrt{2^n}} \sum_{z\in\{0,1\}^n} (-1)^{x\cdot z}|z\rangle. \tag{7.3}$$

Plugging this into Eq. (7.2), the final state of the algorithm before measurement is

$$\begin{aligned} \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} (-1)^{f(x)} H^{\otimes n}|x\rangle &= \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} (-1)^{f(x)} \frac{1}{\sqrt{2^n}} \sum_{z\in\{0,1\}^n} (-1)^{x\cdot z}|z\rangle \\ &= \sum_{z\in\{0,1\}^n} \left(\frac{1}{2^n} \sum_{x\in\{0,1\}^n} (-1)^{f(x)+x\cdot z} \right) |z\rangle. \end{aligned} \tag{7.4}$$

To see how measuring this state lets us determine whether the function is constant or balanced, let us calculate the probability of getting all zeros $|0\ldots00\rangle$. The amplitude of $|0\ldots00\rangle$ (right before measurement) is

$$\frac{1}{2^n}\sum_{x\in\{0,1\}^n}(-1)^{f(x)}.$$

This amplitude depends on whether $f(x)$ is constant or balanced:

- If $f(x)$ is constant, then $f(x)$ always outputs the same value, so $f(x) = f(0\ldots00)$ for all x, and the amplitude is

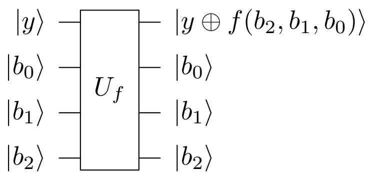
  $$\frac{1}{2^n}\sum_{x\in\{0,1\}^n}(-1)^{f(0\ldots00)} = \frac{1}{2^n}(-1)^{f(0\ldots00)}2^n = (-1)^{f(0\ldots00)}.$$

  Taking the norm-square of this, if $f(x)$ is constant, the probability of measuring $|0\ldots00\rangle$ is 1.
- If $f(x)$ is balanced, then $(-1)^{f(x)}$ is 1 half the time and -1 the other half the time, so the amplitude is 0. Hence, if $f(x)$ is balanced, the probability of measuring $|0\ldots00\rangle$ is 0, so we are guaranteed to get something other than $|0\ldots00\rangle$ when we measure.

Thus, to determine if f is constant or balanced, we measure the n qubits, and if we get $|0\ldots00\rangle$, the function is constant, and if we get anything else, the function is balanced.

Exercise 7.11. Apply $H\otimes H\otimes H$ to $|000\rangle$, and show that the resulting state is a uniform superposition of all binary strings of length 3. If you measure the qubits, what possible outcomes can you get, and with what probabilities?

Exercise 7.12. There is a function on three bits $f(b_2, b_1, b_0)$, with the promise that the function is constant or balanced. You want to determine which by querying an oracle. Go to `https://bit.ly/38P0Nig`. By following this link, you should have access to a custom gate called U_f. This is the oracle, and it acts on four qubits by

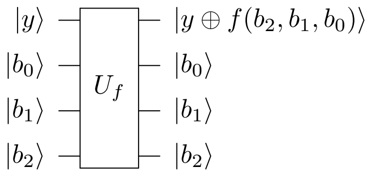


(a) In Quirk, use the Deutsch-Jozsa algorithm to determine whether $f(x)$ is constant or balanced using just one query to U_f. Note you will need to prepare the answer qubit so that it is in the minus state.
(b) In Quirk, query U_f in various ways to determine $f(b_2, b_1, b_0)$:

b_2	b_1	b_0	$f(b_2,b_1,b_0)$
0	0	0	?
0	0	1	?
0	1	0	?
0	1	1	?
1	0	0	?
1	0	1	?
1	1	0	?
1	1	1	?

(c) Since you now know $f(b_2,b_1,b_0)$ completely from part (b), verify that it agrees with your result from part (a).
(d) In the worst case, how many queries does it take to solve the problem classically?

Exercise 7.13. There is a function on three bits $f(b_2,b_1,b_0)$, with the promise that the function is constant or balanced. You want to determine which by querying an oracle. Go to `https://ibm.co/3EWbltg`. By following this link, you should have access to a custom gate called U_f. This is the oracle, and it acts on four qubits by



(a) Program the Deutsch-Jozsa algorithm in IBM Quantum, and use the quantum simulator to determine if $f(b_0,b_1,b_2)$ is constant or balanced.
(b) Run the circuit on an actual quantum processor using IBM Quantum. Which processor did you use, and what histogram of results do you get?

7.4 Secret Dot Product String

7.4.1 The Problem

Deutsch's algorithm and the Deutsch-Jozsa algorithm both followed the same steps: apply Hadamard gate(s), query the oracle, apply Hadamard gate(s) again, and then measure. Since this worked so well, are there any other problems that can be solved by this procedure?

The answer is yes. There is another problem that a quantum computer can solve using this procedure, and it is finding a secret n-bit string by querying an oracle that takes the dot product of the string with the input. That is, we again have a function f that takes as input a binary string of length n and outputs 0 or 1, so $f:\{0,1\}^n \to \{0,1\}$. But now the promise is that $f(x) = s \cdot x$, where s is some n-bit string $s_{n-1}\dots s_1 s_0$, and the dot product of s and x is the sum of the products of their elements, i.e.,

$$s \cdot x = s_{n-1}x_{n-1} + \cdots + s_1 x_1 + s_0 x_0.$$

The problem is to find s, which means finding $s_{n-1}\dots s_1 s_0$.

7.4.2 Classical Solution

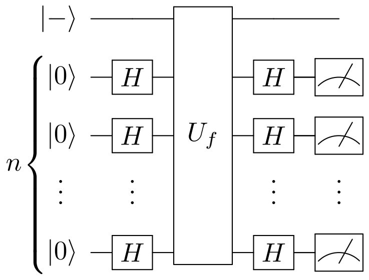
Since we need to determine all n bits of s, the classical solution requires n queries, one to learn each bit of s. For example, if $n = 4$, then

$$\begin{aligned} f(0001) &= s_3(0) + s_2(0) + s_1(0) + s_0(1) = s_0, \\ f(0010) &= s_3(0) + s_2(0) + s_1(1) + s_0(0) = s_1, \\ f(0100) &= s_3(0) + s_2(1) + s_1(0) + s_0(0) = s_2, \\ f(1000) &= s_3(1) + s_2(0) + s_1(0) + s_0(0) = s_3. \end{aligned}$$

It is known that a bounded-error probabilistic algorithm must also take at least n queries to f, but the details are beyond the scope of this textbook.

7.4.3 Quantum Solution: Bernstein-Vazirani Algorithm

Quantumly, we only need one query using the *Bernstein-Vazirani algorithm*, which is a polynomial speedup over classical computers. It follows the exact same steps as the Deutsch-Jozsa algorithm, where we apply Hadamards, query the oracle, and then apply Hadamards again, so as a quantum circuit, it is



Let us work out the math to show that this works. Ignoring the answer qubit, from Eq. (7.4), the state of the n qubits before measurement is

$$\sum_{z \in \{0,1\}^n} \left(\frac{1}{2^n} \sum_{x \in \{0,1\}^n} (-1)^{f(x) + x \cdot z} \right) |z\rangle.$$

For the problem of the secret dot product string, $f(x) = s \cdot x$. Plugging this in, we get

$$\sum_{z \in \{0,1\}^n} \left(\frac{1}{2^n} \sum_{x \in \{0,1\}^n} (-1)^{(s+z) \cdot x} \right) |z\rangle.$$

where $s + z$ denotes bitwise addition (no carry), also known as bitwise XOR. That is, $(s+z)_i = s_i \oplus z_i$. Now we measure this, and to determine the possible measurement outcomes, let us consider the amplitude of getting $|s\rangle$. When $z = s$, $s + z$ is a bit string of all zeros. Then the amplitude of $|s\rangle$ is

$$\frac{1}{2^n}\sum_x(-1)^0 = \frac{1}{2^n}\sum_x 1 = \frac{1}{2^n}2^n = 1.$$

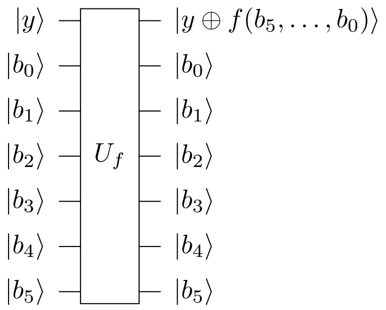
Thus, normalization implies that the amplitude of all other states is 0, so the final state of the qubits is

$$|s\rangle.$$

Measuring this is certain to yield $|s\rangle$, and we have determined s with just one query to the oracle.

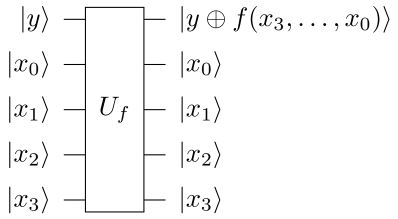
This is a polynomial speedup over the $O(n)$ queries needed by a classical computer. The above speedup holds for bounded error, so it yields an oracle separation between the complexity classes P and BQP. However, the problem is efficient for both classical and quantum computers. The next algorithm, Simon's algorithm, gives the first "true" exponential speedup, where the problem is inefficient for a classical computer, but efficient for a quantum computer, in the number of oracle queries.

Exercise 7.14. There is a function on six bits $f(b_5,b_4,b_3,b_2,b_1,b_0) = s_5b_5 + \cdots + s_1b_1 + s_0b_0$. Find $s = s_5\ldots s_1s_0$ by querying an oracle. Go to `https://bit.ly/31YCBZu`. By following this link, you should have access to a custom gate called U_f. This is the oracle, and it acts on seven qubits by



(a) In Quirk, use the Bernstein-Vazirani algorithm to determine s using just one query to U_f. Note you will need to prepare the answer qubit so that it is in the minus state.
(b) In Quirk, query U_f in various ways to determine each bit of s. Verify that it agrees with your result from part (a).
(c) In the worst case, how many queries does it take to solve the problem classically?

Exercise 7.15. There is a function on four bits $f(x_3,x_2,x_1,x_0) = s_3x_3 + s_2x_2 + s_1x_1 + s_0x_0$. Find $s = s_3s_2s_1s_0$ by querying an oracle. Go to `https://ibm.co/3INITfq`. By following this link, you should have access to a custom gate called U_f. This is the oracle, and it acts on five qubits by



(a) Program the Bernstein-Vazirani algorithm in IBM Quantum, and use the quantum simulator to find $s = s_3s_2s_1s_0$.
(b) Run the circuit on an actual quantum processor using IBM Quantum. Which processor did you use, and what histogram of results do you get?

Exercise 7.16. In the Bernstein-Vazirani algorithm, recall the final state of the qubits (before measurement) is

$$\frac{1}{2^n} \sum_{z\in\{0,1\}^n} \left(\sum_{x\in\{0,1\}^n} (-1)^{(s+z)\cdot x} \right) |z\rangle.$$

Say $n = 3$ and consider $z \neq s$ such that $s+z = 001$ (using bitwise addition). Show that the amplitude of this choice of $|z\rangle$ is zero by filling in the following table, and then computing the sum of the last column.

x	$(s+z)\cdot x$	$(-1)^{(s+z)\cdot x}$
000	?	?
001	?	?
010	?	?
011	?	?
100	?	?
101	?	?
110	?	?
111	?	?
$\sum_x (-1)^{(s+z)\cdot x}$:		?

7.4.4 Recursive Problem

The problem of finding a hidden dot product string can be made recursive, meaning we embed the problem in a bigger instance of the problem, which is embedded in a bigger instance of the problem, and so forth. The details are beyond the scope of this textbook, but if we have k levels, then a classical computer takes $\Omega(n^k)$ queries to solve the problem (recall from Section 1.7.1 that big-Ω is a lower bound, so a classical computer takes at least this many queries). In contrast, a quantum computer using a recursive version of the Bernstein-Vazirani algorithm, however, can solve this problem with 2^k queries. If we have $k = \log_2(n)$ levels, then the classical algorithm takes $\Omega(n^{\log n})$ queries, which is bigger than any polynomial. We call this *superpolynomial*. The quantum computer, however, only takes n queries, which is linear and efficient. Thus, the recursive version of the hidden dot product string problem shows that a quantum computer can yield a superpolynomial speedup in queries. This speedup, however, is less than exponential. Next, we will see a problem with an exponential speedup.

7.5 Secret XOR Mask

7.5.1 The Problem

In this problem, the oracle takes as input an n-bit string $x = x_{n-1} \dots x_1 x_0$ and outputs an n-bit string $f(x) = f_{n-1} \dots f_1 f_0$. That is, $f : \{0,1\}^n \to \{0,1\}^n$. We are promised that

$$f(x) = f(y)$$

if and only if the two inputs x and y are related by

$$x = y \oplus s, \quad \text{and} \quad y = x \oplus s$$

for some "secret" n-bit string $s = s_{n-1} \dots s_1 s_0 \neq 0 \dots 00$, where $\oplus$ denotes the bitwise XOR. That is, $f(x) = f(y)$ if and only if

$$x_i = y_i \oplus s_i, \quad \text{and} \quad y_i = x_i \oplus s_i.$$

The goal is to find the secret n-bit string $s = s_{n-1} \dots s_1 s_0$. The secret bit string is called a *mask*, and since it is used to XOR the inputs, it is called an *XOR mask*. The problem is to find the secret XOR mask $s = s_{n-1} \dots s_1 s_0$.

For example, say $n = 3$ and $s = 110$. Then, for each value of x, $x \oplus s$ is shown in the following table:

x	$x \oplus s$
000	110
001	111
010	100
011	101
100	010
101	011
110	000
111	001

Notice these come in pairs. That is, 000 and 110 are a pair, 001 and 111 are a pair, 010 and 100 are a pair, and 011 and 101 are a pair. This is because if $y = x \oplus s$, then it is automatically true that $x = y \oplus s$. As a proof, we start with

$$y = x \oplus s.$$

Next, if we XOR both sides with s, we get

$$y \oplus s = x \oplus s \oplus s.$$

Since $s \oplus s = 0$, this is

$$y \oplus s = x \oplus 0.$$

Thus,

$$y \oplus s = x,$$

or reversing the two sides, $x = y \oplus s$. Now, from the promise about the oracle, for each pair x and y, $f(x)$ and $f(y)$ must be the same. For example, here are two possible truth tables for $f(x)$, satisfying that $f(x) = f(y)$ if and only if $y = x \oplus s$:

x	$f(x)$
000	011
001	101
010	001
011	000
100	001
101	000
110	011
111	101

x	$f(x)$
000	110
001	001
010	111
011	000
100	111
101	000
110	110
111	001

Notice that in both examples, $f(000) = f(110)$, $f(001) = f(111)$, $f(010) = f(100)$, and $f(011) = f(101)$. Also note there are 1680 different possible truth tables for $f(x)$. This is because we have four pairs that we need to assign outputs to, and there are $2^3 = 8$ different outputs. For the first pair, we have 8 choices of outputs. For the second pair, we have 7 choices of outputs. For the third pair, we have 6 choices of outputs. And for the fourth pair, we have 5 choices of outputs. Altogether, we have $8 \cdot 7 \cdot 6 \cdot 5 = 1680$ possible permutations.

Exercise 7.17. Say $n = 3$ and $s = 010$.
(a) Find the pairs of n-bit strings x and y such that $y = x \oplus s$.
(b) Give a possible truth table for $f(x)$ that satisfies the promise that $f(x) = f(y)$ if and only if $y = x \oplus s$.

7.5.2 Classical Solution

Classically, we can find the secret XOR mask s by finding a *collision*, meaning a pair x and y such that f maps them to the same string, i.e., $f(x) = f(y)$. From the promise about f, this implies that $x = y \oplus s$ and $y = x \oplus s$, and we can take the XOR of x and y to find s:

$$x \oplus y = x \oplus (x \oplus s) = \underbrace{(x \oplus x)}_{0} \oplus s = s.$$

One approach is to trying the inputs one-by-one until we find a collision. In the worst case, we could try half of the inputs without yet seeing a collision. We are guaranteed, however, that trying one more input will yield a collision, so the query complexity with this approach is $O(2^{n-1} + 1)$.

We can do better, however. If we query f with *random* inputs. This prevents f from being designed to be as worse as possible as previously described, where half the inputs, plus 1, must be queried to find a collision. Now, say we have queried

f a total of k times, so we have k values of f. The probability of there being a collision in these k values of f is given by the number of pairs of values, which is the combination ${}_kC_2 = k(k-1)/2 = O(k^2)$. Since this grows quadratically with the number of queries, one expects to query f roughly $\sqrt{2^n} = 2^{n/2}$ times in order to find a collision. Although this is an improvement, it is still exponential in n.

Exercise 7.18. We have an oracle $f : \{0,1\}^n \to \{0,1\}^n$ with a secret XOR mask s. Say $n = 4$. Querying the oracle with some various inputs, we find that $f(1011) = 0010$ and $f(0111) = 0010$. What is s?

Exercise 7.19. The task of finding a collision is closely related to a famous problem called the *birthday problem*, which is to find the probability that in a room of n people, at least two of them share the same birthday. We ignore leap years, so there are 365 days in a year. We also assume that people's birthdays are randomly distributed. In reality, this is not true, as some birthdays are more common than others, but this only makes a shared birthday more likely.

To solve this problem, we find the probability that the n people do *not* share any birthdays. Then, the probability that at least two people share the same birthday is 1 minus this. To calculate the probability that no one shares a birthday, we add people to the room one-by-one. The first person in the room does not share a birthday with anyone else because there is no one else. The second person in the room has 364 possible birthdays so as to not share a birthday with the first person, and the probability of this is 364/365. The third person in the room has 363 possible birthdays so as to not share a birthday with the first two people, and the probability of this is 363/365. The fourth person has 362 possible birthdays to avoid sharing, which has a probability of 362/365. Continuing this, the probability of no one sharing a birthday is

$$\frac{364}{365}\frac{363}{365}\frac{362}{365}\cdots\frac{365-(n-1)}{365}.$$

Multiplying this by 365/365, we get

$$\frac{365}{365}\frac{364}{365}\frac{363}{365}\frac{362}{365}\cdots\frac{365-(n-1)}{365} = \frac{365\cdot 364\cdot 363\cdot 362\cdot\ldots\cdot(365-(n-1))}{365^n}.$$

Thus, the probability that at least two people share the same birthday is

$$1-\frac{365\cdot 364\cdot 363\cdot 362\cdot\ldots\cdot(365-(n-1))}{365^n}.$$

This can be calculated using a computer algebra system. For example, with $n = 23$ people,

- Using Mathematica,

```
n=23;
1 - Product[i/365., {i, 365-(n-1), 365}]
```

  The `Product` function multiplies $(365-(n-1))/365$ up through 365/365, and the output of 1 minus this product is 0.507297.

- Using SageMath,

```
sage: n=23
sage: 1 - prod(i/365. for i in ((365-(n-1))..365))
0.507297234323986
```

  The `prod` function multiplies $(365-(n-1))/365$ up through 365/365.

So, there is over a 50% chance that at least two people share the same birthday. This may be higher than one might expect, so some call the birthday problem the *birthday paradox*.

(a) With $n = 30$, what is the probability that at least two of them share the same birthday?
(b) With $n = 40$, what is the probability that at least two of them share the same birthday?
(c) With $n = 50$, what is the probability that at least two of them share the same birthday?
(d) With $n = 60$, what is the probability that at least two of them share the same birthday?

7.5.3 Quantum Solution: Simon's Algorithm

Simon's algorithm follows the pattern we have seen so far: apply Hadamards, query the oracle, and apply Hadamards again. But now we have n input qubits and n answer qubits, and we start each of the answer qubits in the $|0\rangle$ state (so we are using the regular quantum oracle, not the phase oracle). For the oracle, it maps

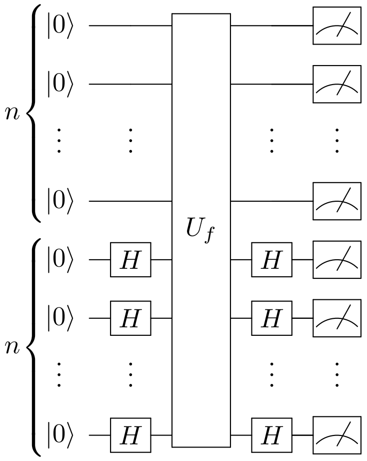
$$|x\rangle|y\rangle \xrightarrow{U_f} |x\rangle|y \oplus f(x)\rangle,$$

where

$$\begin{aligned}
|x\rangle &= |x_{n-1}\dots x_1 x_0\rangle, \\
|y\rangle &= |y_{n-1}\dots y_1 y_0\rangle, \\
|y \oplus f(x)\rangle &= |y_{n-1} \oplus f_{n-1}, \dots, y_1 \oplus f_1, y_0 \oplus f(x_0)\rangle.
\end{aligned}$$

Another difference with Simon's algorithm is that we will measure all the qubits, not just the input qubits.

The quantum circuit for Simon's algorithm is



Let us work through the math of what this does. Initially, we have two n-qubit registers, one for the input qubits, and another for the answer qubits.

$$|0\dots00\rangle|0\dots00\rangle.$$

Now, we apply the Hadamard gate to each of the input qubits, resulting in

$$|+\cdots++\rangle|0\ldots00\rangle.$$

Multiplying out the $|+\rangle$ states, we get a uniform superposition over n-bit strings:

$$\frac{1}{\sqrt{2^n}}\sum_{x\in\{0,1\}^n}|x\rangle|0\ldots00\rangle.$$

Next, querying the oracle, we get

$$\frac{1}{\sqrt{2^n}}\sum_{x\in\{0,1\}^n}|x\rangle|f(x)\rangle.$$

Now, we again apply the Hadamard gate to each of the input qubits, resulting in

$$\frac{1}{\sqrt{2^n}}\sum_{x\in\{0,1\}^n}H^{\otimes n}|x\rangle|f(x)\rangle.$$

From Eq. (7.3), $H^{\otimes n}|x\rangle$ is a uniform superposition of bit strings $|z\rangle$ multiplied by a phase of $(-1)^{x\cdot z}$, so we get

$$\frac{1}{\sqrt{2^n}}\sum_{x\in\{0,1\}^n}\frac{1}{\sqrt{2^n}}\sum_{z\in\{0,1\}^n}(-1)^{x\cdot z}|z\rangle|f(x)\rangle.$$

Now, let us measure the answer qubits. We will get one particular value of $f(x)$. Let us call the value f'. There are two values of x for which $f(x)=f'$. Let us call them x' and x''. That is, $f(x')=f(x'')=f'$. So, x' and x'' are a pair of inputs for which there is a collision. Then, the state will collapse to these two values of x:

$$\frac{1}{\sqrt{2}}\frac{1}{\sqrt{2^n}}\sum_{z\in\{0,1\}^n}\left[(-1)^{x'\cdot z}+(-1)^{x''\cdot z}\right]|z\rangle|f'\rangle.$$

Note the first coefficient went from $1/\sqrt{2^n}$ to $1/\sqrt{2}$ because the number of possible outcomes for x went from 2^n (all possible bit strings) to 2 ($|x'\rangle$ and $|x''\rangle$). Combining the coefficients,

$$\frac{1}{\sqrt{2^{n+1}}}\sum_{z\in\{0,1\}^n}\left[(-1)^{x'\cdot z}+(-1)^{x''\cdot z}\right]|z\rangle|f'\rangle.$$

Next, we measure the input qubits. To determine the possible results, note that $(-1)^{x'\cdot z}=\pm1$ and $(-1)^{x''\cdot z}=\pm1$ depending on what x' and x'' are. Then, their sum is either ±2 or 0:

$$(-1)^{x'\cdot z}+(-1)^{x''\cdot z}=\begin{cases}\pm2, & x'\cdot z=x''\cdot z \bmod 2,\\ 0, & x'\cdot z\neq x''\cdot z \bmod 2.\end{cases}$$

Thus, when measuring the input qubits, we only get a value of $|z\rangle$ where

$$x' \cdot z = x'' \cdot z \bmod 2.$$

Adding $x'' \cdot z$ to both sides, we get

$$x' \cdot z + x'' \cdot z = x'' \cdot z + x'' \cdot z \bmod 2.$$

The right-hand side of this equation is 0 because if we add any bit to itself modulo 2, we get 0. Thus,

$$x' \cdot z + x'' \cdot z = 0 \bmod 2.$$

Factoring the left-hand side,

$$(x' + x'') \cdot z = 0 \bmod 2.$$

Since x' and x'' are a pair of inputs for which there is a collision, $x' \oplus x'' = s$. Then, we have

$$s \cdot z = 0 \bmod 2.$$

Thus, when measuring the input qubits, we get a value of $|z\rangle = |z_{n-1} \dots z_1 z_0\rangle$ such that its dot product with s is 0 mod 2. Writing out the dot product,

$$s_{n-1}z_{n-1} + \cdots + s_1 z_1 + s_0 z_0 = 0 \bmod 2. \tag{7.5}$$

This is an equation containing all n of our unknowns, the s_i's.

If we repeat this process, we will get a $|z\rangle = |z_{n-1} \dots z_1 z_0\rangle$ that satisfies Eq. (7.5) and is likely different from the first because there are an exponential number of them. To see this, the probability of measuring any such $|z\rangle$ is

$$\left| \frac{\pm 2}{\sqrt{2^{n+1}}} \right|^2 = \frac{4}{2^{n+1}} = \frac{1}{2^{n-1}}.$$

Or, put another way, there are 2^{n-1} possible $|z\rangle$'s whose dot product with s is zero, and we have the same probability of getting each one.

Repeating the quantum algorithm $O(n)$ times, we can get n different $|z\rangle$'s, each satisfying Eq. (7.5). Together, they are a system of n equations and n unknowns, which we can solve for the s_i's. Thus, we can find s with $O(n)$ queries to the oracle, and this was the first exponential oracle separation between classical and quantum computers.

Exercise 7.20. You are using Simon's algorithm to find an $n = 3$ bit string $s = s_2 s_1 s_0$. You run the quantum circuit three times, and you get the following values for $|z\rangle$, such that $s \cdot z = 0 \bmod 2$:

$$|001\rangle, |110\rangle, |111\rangle,$$

What is s?

Exercise 7.21. You are using Simon's algorithm to find an $n = 8$ bit string $s = s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0$.
(a) How many different values of $|z\rangle$ are there, such that $s \cdot z = 0 \bmod 2$?

(b) If you run the quantum circuit three times, what is the probability that all three values of $|z\rangle$ are different?

7.5.4 Summary

We have examined several quantum algorithms that all follow the same procedure: apply Hadamards, query the oracle, and apply Hadamards again. The following table summarizes the problems, query complexities, and asymptotic quantum speedups:

Problem	Classical Queries	Quantum Algorithm	Quantum Queries	Asymptotic Speedup
n-bit Parity	n	Deutsch	$n/2$	None
Constant vs Balanced	Exact: $2^{n-1}+1$ Bounded: $O(1)$	Deutsch-Jozsa	1	Exponential None
Dot Product String	n	Bernstein-Vazirani	1	Polynomial
Recursive Dot Product String	$\Omega(n^{\log_2 n})$	Recursive Bernstein-Vazirani	n	Superpolynomial
XOR Mask	$O(2^{n/2})$	Simon	$O(n)$	Exponential

We started with the parity problem. The quantum algorithm does offer an improvement in that it takes half as many queries, but asymptotically, both the classical and quantum algorithms are $O(n)$, so there is no speedup in that sense. Then, we looked at determining whether the oracle is constant or balanced. Although the quantum algorithm yields an exponential improvement over the exact classical algorithm, it is no improvement over the bounded algorithm that is often acceptable in practice. For a true asymptotic speedup, the problem of finding a secret dot product string is solved by a quantum computer using polynomially fewer queries, and a recursive version of the problem is solved with superpolynomially fewer queries. Finally, finding a secret XOR mask takes exponentially fewer queries on a quantum computer, which shows that for oracular problems, quantum computers can yield an exponential speedup.

7.6 Brute-Force Searching

7.6.1 The Problem

Before moving on to problems where we can calculate the circuit complexity, let us discuss one more oracular problem where we count the number of oracle queries.